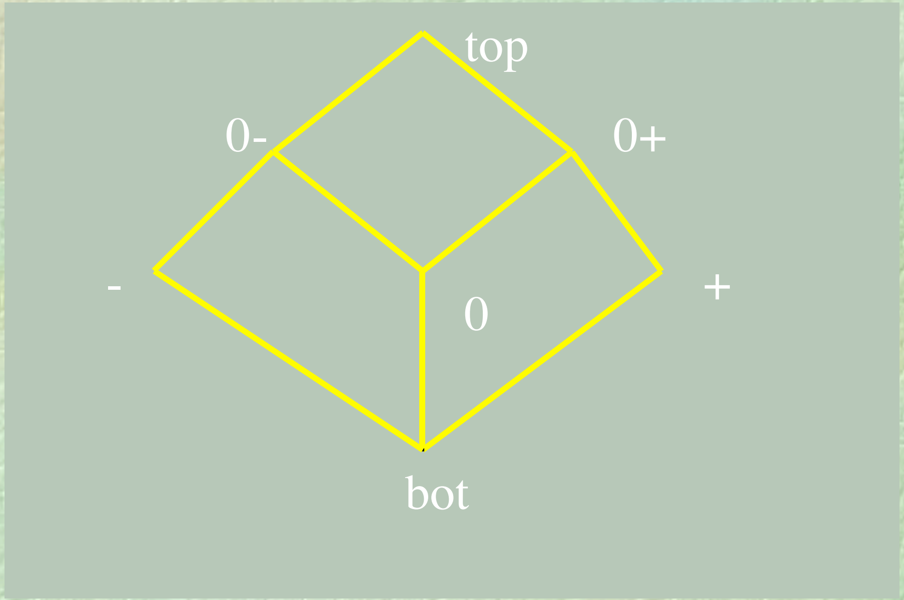
A simple abstract interpreter to compute *Signs*

The `Sign` Abstract Domain

- concrete domain ($\mathcal{P}(\mathcal{Z})$, $\subseteq$, $\emptyset$, $\mathcal{Z}$, $\cup$, $\cap$)
  sets of integers
- abstract domain (*Sign*, $\leq$, *bot, top, lub, glb*)



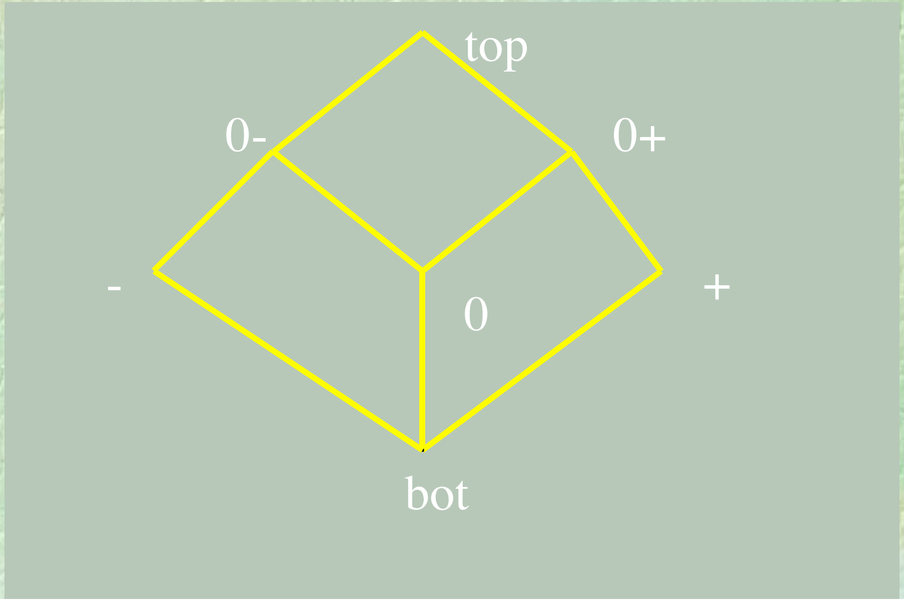
The example of `Sign`

$\gamma_{\texttt{Sign}}(x) =$

- $\emptyset$, if x= bot
- $\{y|y>0\}$, if x= +
- $\{y|y\geq 0\}$, if x= 0+
- $\{0\}$, if x= 0
- $\{y|y\leq 0\}$, if x= 0-
- $\{y|y<0\}$, if x= -
- $\mathcal{Z}$, if x= top

$\alpha_{\texttt{Sign}}(y) =$ glb of

- bot , if y= $\emptyset$
- - , if y $\subseteq \{y|y<0\}$
- 0- , if y $\subseteq \{y|y\leq 0\}$
- 0 , if y = $\{0\}$
- 0+ , if y $\subseteq \{y|y \geq 0\}$
- + , if y $\subseteq \{y|y>0\}$
- top , if y $\subseteq \mathcal{Z}$



A simple abstract interpreter computing *Signs*

- concrete semantics
  - executable specification (in ML) of the denotational semantics of untyped λ-calculus without recursion
- abstract semantics
  - abstract interpreter computing on the domain *Sign*

The language: syntax

- type ide = Id of string
- type exp =

  ```
  | Eint of int
  | Var of ide
  | Times of exp * exp
  | Ifthenelse of exp * exp * exp
  | Fun of ide * exp
  | Appl of exp * exp
  ```

A program

```
Fun(Id "x",
        Ifthenelse(Var (Id "x"),
        Times (Var (Id "x"), Var (Id "x")),
                        Times (Var (Id "x"), Eint (-1))))
```

- the ML expression

```
function x -> if x=0 then x * x else x * (-1)
```

Concrete semantics

- denotational interpreter
- eager semantics
- separation from the main semantic evaluation function of the primitive operations
  - which will then be replaced by their abstract versions
- abstraction of concrete values
  - identity function in the concrete semantics
- symbolic “non-deterministic” semantics of the conditional

Semantic domains

-
  ```
  type eval =
  | Funval of (eval -> eval)
  | Int of int
  | Wrong
  ```

-
  ```
  let alfa x = x
  ```

-
  ```
  type env = ide -> eval
  ```

```
let emptyenv (x: ide) = alfa(Wrong)
let applyenv ((x: env), (y: ide)) = x y
let bind ((r:env), (l:ide), (e:eval)) (lu:ide) =
 if lu = l then e else r(lu)
```

Semantic evaluation function

- let rec sem (e:exp) (r:env) = match e with

```
| Eint(n) -> alfa(Int(n))
| Var(i) -> applyenv(r,i)
| Times(a,b) ->  times ( (sem a r), (sem b r))
| Ifthenelse(a,b,c) -> let a1 = sem a r in
                        (if valid(a1) then sem b r else
                        (if unsatisfiable(a1) then sem c r
                        else merge(a1,sem b r,sem c r)))
| Fun(ii,aa) ->  makefun(ii,aa,r)
| Appl(a,b) -> applyfun(sem a r, sem b r)
```

Primitive operations

```
let times (x,y) = match (x,y) with
        |(Int nx, Int ny) -> Int (nx * ny)
        | _ -> alfa(Wrong)

let valid x = match x with
            |Int n -> n=0

let unsatisfiable x = match x with
            |Int n -> not n=0

let merge (a,b,c) = match a with
             |Int n -> if b=c then b else alfa(Wrong)
             | _ -> alfa(Wrong)

let applyfun ((x:eval),(y:eval)) = match x with
             |Funval f -> f y
             | _ -> alfa(Wrong)

let rec makefun(ii,aa,r) = Funval(function d ->
            if d = alfa(Wrong) then alfa(Wrong)
            else sem aa (bind(r,ii,d)))
```

From the concrete to the collecting semantics

- the concrete semantic evaluation function
  - sem: exp -> env -> eval
- the collecting semantic evaluation function
  - semc: exp -> env -> Π(eval)
  - semc e r = {sem e r}
  - all the concrete primitive operations have to be lifted to Π(eval) in the design of the abstract operations

Example of concrete evaluation

```
# let esempio = sem(
  Fun
     (Id "x",
      Ifthenelse
       (Var (Id "x"), Times (Var (Id "x"), Var (Id "x")),
        Times (Var (Id "x"), Eint (-1)))) )    emptyenv;;
val esempio : eval = Funval <fun>

# applyfun(esempio,Int 0);;
- : eval = Int 0
# applyfun(esempio,Int 1);;
- : eval = Int -1
# applyfun(esempio,Int(-1));;
- : eval = Int 1
```

- in the "virtual" collecting version

  applyfunc(esempio,{Int 0,Int 1}) = {Int 0, Int -1}

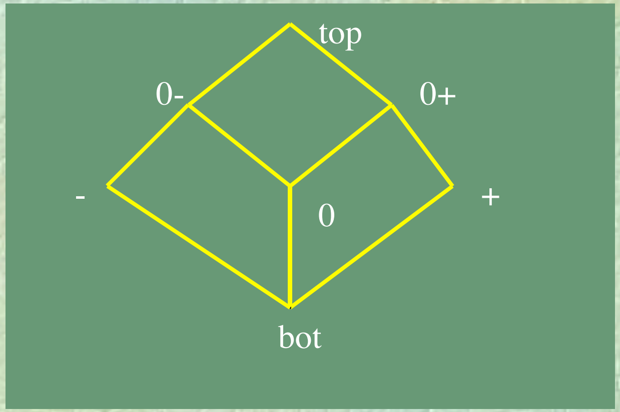
  applyfunc(esempio,{Int 0,Int -1}) = {Int 0, Int 1}

  applyfunc(esempio,{Int -1,Int 1}) = {Int 1, Int -1}

From the collecting to the abstract semantics

- concrete domain: $(\Pi(\text{ceval}), \subseteq)$
- concrete (non-collecting) environment:
  - cenv = ide -> ceval
- abstract domain: $(\text{eval}, \leq)$
- abstract environment: env = ide -> eval
- the collecting semantic evaluation function
  - semc: exp -> env -> Π(ceval)
- the abstract semantic evaluation function
  - sem: exp -> env -> eval

The *Sign* Abstract Domain



- concrete domain ($\mathcal{P}(\mathcal{Z})$, $\subseteq$, $\emptyset$, $\mathcal{Z}$, $\cup$, $\cap$)

  sets of integers
- abstract domain (*Sign*, ≤, *bot*, *top*, *lub*, *glb*)

Redefining eval for *Sign*

```
type ceval = Funval of (ceval -> ceval) | Int of int | Wrong
type eval =  Afunval of (eval -> eval) | Top | Bottom | Zero | Zerop | Zerom | P | M

let alfa x = match x with    Wrong -> Top
               | Int n -> if n = 0 then Zero else if n > 0 then P else M
```

- the partial order relation ≤
  - the relation shown in the *Sign* lattice, extended with its lifting to functions
    - there exist no infinite increasing chains
    - we might add a recursive function construct and find a way to compute the abstract least fixpoint in a finite number of steps
- lub and glb of eval are the obvious ones
- concrete domain: ($\mathcal{P}$(ceval), ⊆, ∅, ceval, ∪, ∩)
- abstract domain: (eval, ≤, Bottom, Top, lub, glb)

Concretization function

- concrete domain: $(\mathcal{P}(ceval), \subseteq, \emptyset, ceval, \cup, \cap)$
- abstract domain: $(eval, \leq, Bottom, Top, lub, glb)$
- $\gamma_s(x) =$

$$
\begin{array}{ll}
\{\}, & \text{if } x = \text{Bottom} \\
\{Int(y) \mid y>0\}, & \text{if } x = \text{P} \\
\{Int(y) \mid y\geq 0\}, & \text{if } x = \text{Zerop} \\
\{Int(0)\}, & \text{if } x = \text{Zero} \\
\{Int(y) \mid y\leq 0\}, & \text{if } x = \text{Zerom} \\
\{Int(y) \mid y<0\}, & \text{if } x = \text{M} \\
ceval, & \text{if } x = \text{Top} \\
\{Funval(g) \mid \forall y \in eval\ \forall x \in \gamma_s(y), & \\
\quad g(x) \in \gamma_s(f(y))\}, & \text{if } x = \text{Afunval}(f)
\end{array}
$$

Abstraction function

- concrete domain: ($\mathcal{P}$(ceval), $\subseteq$, $\emptyset$, ceval, $\cup$, $\cap$)
- abstract domain: (eval, $\leq$, Bottom, Top, lub, glb)
- α_s(y) = glb{

Bottom, if y = {}

M, if y $\subseteq$ {Int(z)| z<0}

Zerom, if y $\subseteq$ {Int(z)| z$\leq$0}

Zero, if y = {Int(0)}

Zerop, if y $\subseteq$ {Int(z)| z $\geq$ 0}

P, if y $\subseteq$ {Int(z)| z>0}

Top, if y $\subseteq$ ceval

lub{Afunval(f)| Funval(g) $\in$ γ_s(Afunval(f))},
if y $\subseteq$ {Funval(g)} & Funval(g) $\in$ y}}

Galois connection

- α_s and γ_s
  - are monotonic
  - define a Galois connection

Times$^{\text{Sign}}$

	bot	-	0-	0	0+	+	top
bot	bot	bot	bot	bot	bot	bot	bot
-	bot	+	0+	0	0-	-	top
0-	bot	0+	0+	0	0-	0-	top
0	bot	0	0	0	0	0	0
0+	bot	0-	0-	0	0+	0+	top
+	bot	-	0-	0	0+	+	top
top	bot	top	top	0	top	top	top

- optimal (hence correct) and complete (no approximation)

Abstract operations

- in addition to times and lub

```
let valid x = match x with
     | Zero -> true
     | _ -> false
let unsatisfiable x = match x with
     | M ->  true
     | P -> true
     | _ -> false
let merge (a,b,c) = match a with
     | Afunval(_) -> Top
     | _ -> lub(b,c)
let applyfun ((x:eval),(y:eval)) = match x with
     |Afunval f -> f y
     | _ -> alfa(Wrong)
let rec makefun(ii,aa,r) = Afunval(function d ->
     if d = alfa(Wrong) then d else sem aa (bind(r,ii,d)))
```

- sem is left unchanged

An example of abstract evaluation

```
# let esempio = sem(
 Fun
    (Id "x",
     Ifthenelse
      (Var (Id "x"), Times (Var (Id "x"), Var (Id "x")),
       Times (Var (Id "x"), Eint (-1)))) )    emptyenv;;
val esempio : eval = Afunval <fun>
```

```
applyfunc(esempio,{Int 0,Int 1}) = {Int 0, Int -1}
applyfunc(esempio,{Int 0,Int -1}) = {Int 0, Int 1}
applyfunc(esempio,{Int -1,Int 1}) = {Int 1, Int -1}
```

```
# applyfun(esempio,P);;
- : eval = M
# applyfun(esempio,Zero);;
- : eval = Zero
# applyfun(esempio,M);;
- : eval = P
# applyfun(esempio,Zerop);;
- : eval = Top
# applyfun(esempio,Zerom);;
- : eval = Zerop
# applyfun(esempio,Top);;
- : eval = Top
```

- wrt the abstraction of the concrete (collecting) semantics, approximation for Zerop
- no abstract operations which “invent” the values Zerop and Zerom
    - which are the only ones on which the conditional takes both ways and can introduce approximation

Recursion

- the language has no recursion
  - fixpoint computations are not needed
- if (sets of) functions on the concrete domain are abstracted to functions on the abstract domain, we must be careful in the case of recursive definitions
  - a naïve solution might cause the application of a recursive abstract function to diverge, even if the domain is finite
  - we might never get rid of recursion because the guard in the conditional is not valid or satisfiable
  - we cannot explicitely compute the fixpoint, because equivalence on functions cannot be expressed
  - termination can only be obtained by a loop checking mechanism (finitely many different recursive calls)
- we will see a different solution in a case where (sets of) functions are abstracted to non functional values
  - the explicit fixpoint computation will then be possible